

Locating Faults in AspectJ Programs

Sai Zhang, Jianjun Zhao
School of Software
Shanghai Jiao Tong University
800 Dongchuan Road, Shanghai 200240, China
saizhang,zhao-jj@sjtu.edu.cn

ABSTRACT

As Aspect-Oriented Programming (AOP) wins more and more popularity, there is increasing interest in using aspects to implement crosscutting concerns in object-oriented software. During software evolution, source code editing and testing are interleaved activities to assure code quality. When regression tests fail unexpectedly after a long session of editing, it may be difficult for programmers to find out the failure causes. Moreover, due to the extensive use of subtyping and dynamic dispatch in object-oriented programming language and the inherent semantic intricacies in aspect-oriented features, the nontrivial combination of those likely non-local changes would also affect other parts of program accidentally and that may indicate potential faults in the updated version.

In this paper, we propose a new fault locating technique for AspectJ programs. At the core of our approach is the source level *atomic change* representation, which captures precisely the semantic differences between two program versions. If a test fails unexpectedly, a subset of affecting changes is identified and programmers are allowed to select (and apply) suspected atomic changes to the original program, constructing compliant intermediate versions. Then programmers can re-execute the failed test against these intermediate program versions to locate the exact failure-inducing reasons by iteratively select, apply and narrow down the set of affecting changes. We implement our prototype Flota, a fault locating tool for AspectJ programs built on top Celadon¹. Our empirical study shows that Flota can help programmers effectively find a small set of failure-inducing changes and provide valuable debugging support for AspectJ programs.

1. INTRODUCTION

During software development process, coding and testing are interleaved activities to assure code quality. Normally, when new program functionality is implemented, or an existing program is modified, the updated software version needs to be regress tested to validate these changes. After a long code editing session, regression tests are executed to ensure that the changed code in the updated program version does not conflict with previous releases. In this phase, any test case that produces unexpected result may indicate potential defects in the updated software. Difficulties occur when regression tests reveal unexpected behaviors, such as assertion failure or exceptions. Sometimes, although the programmer knows that he has introduced a bug, he still does not know which part of editing should be responsible for the bug. If the editing are trivially small, it may be easy to find the buggy code by inspecting all these changed places manually. However, as a code base and its test suite grow in size, running the tests after each minor change

become infeasible, and the number of changes between successive executions of the test suite is likely to increase. In such cases, examining each place of source modification and pinpointing the few that introduces the failure become a burden task for programmers. Moreover, edits are inter-related in many ways especially when developing large software system, and there can be more than one changes that should be responsible for the failed tests.

Aspect-Oriented Programming (AOP) [11] has been proposed as a technique for improving separation of concerns in software design and implementation. In an aspect-oriented system, the basic program unit is an aspect rather than a procedure or a class. An aspect with its encapsulation of state with associated advice is a significant different abstraction than the procedure unit within procedural programs or the class unit within object-oriented programs. With the inclusion of *join point*, an aspect woven into the base code is solely responsible for a particular crosscutting concern, which raises the system's modularity among aspects and classes. However, when locating faulty code in aspect-oriented programs, it involves more complex situations than in the traditional programming languages:

- **Control flow and data dependence between aspect and base code.** Since the woven aspect may change control and data dependency to the base code, adding or changing the aspect code can significantly affect the semantics of whole program.
- **Multiple Advice Invocation.** While multiple advices apply at the same join point, *precedence rules* [3] determine the order in which they execute. The interactions between the base and aspect code or even the aspect weaving sequences will also dramatically affect the program behavior.
- **Advice with dynamic pointcuts.** For an advice associated with a dynamic pointcut, the advice may or may not be invoked at a join point at run time, depending on the evaluation of the corresponding runtime condition.
- **After advices and exception handling.** The invocation of *after advices* is more complex, because it is related to the exception handling mechanism in AspectJ. *After advices* are classified into three types: *after-returning*, *after-throwing* and *after-always*, which have different semantics in program understanding.

Although many fault localization techniques have been presented in the literature, most of the work has been focused on procedural or object-oriented software [7, 8, 19, 22], seldom effort has been made for aspect-oriented software. The unique aspectual features in AspectJ programs make it more difficult to identify the failure causes of AspectJ programs and how to debug AspectJ programs still remains a big problem. Therefore, an appropriate fault localization technique which can capture the subtle semantic changes and isolate the faulty changes for AspectJ programs is needed. In this

¹Our change impact analysis framework for AspectJ programs [23].

paper, we proposed a new fault localization technique for AspectJ programs. At the core of our approach is the source level *atomic change* representation, which can capture precisely the semantic differences between two AspectJ program versions. We construct static AspectJ call graph for the failed test to identify a subset of affecting changes. Programmers can select and apply (or rollback) the suspected atomic changes to the original program, constructing compliant intermediate versions. Therefore, programmers can re-execute the failed tests against these intermediate program versions to locate the exactly failure-inducing reasons by repeatedly select, apply and then narrow down the set of affecting changes.

We built Flota, a fault localization tool for AspectJ programs on top of the Celadon framework. Flota uses the input of Celadon to generate valid intermediate programs automatically. The benefits of automating the fault localization process relies on: (1) identifying the subset of affecting changes which causes the regression tests fail without examining each of these changes manually, (2) constructing compilable intermediate source program versions, and (3) certain changes that do not result in failure can be ignored, programmers can further examine and isolate smaller sets of changes until they locate the exactly ones. Our goal is to provide programmers with a tool to aid in the debugging process for AspectJ programs, so that they do not need to be concerned with the syntactic inter-relationship of these changes. We present an experimental study on 6 AspectJ program versions [1] using Flota. The result shows that Flota can effectively reduce the number of responsible changes and provide valuable debugging support for AspectJ programs. The main contributions of this paper are:

- We proposed a new fault locating technique for AspectJ programs, based on the *atomic change* [23] representation for AspectJ programming language. We summarized the semantic dependence relationships between these atomic changes, which allow automatically constructing syntactically valid intermediate program versions.
- We implemented Flota, a fault locating tool for AspectJ programs on top of Celadon. Flota can build intermediate program versions automatically according to programmers's selection.
- We conducted an experimental study using 6 AspectJ benchmark versions [1]. The result shows that Flota can help programmers effectively identify fault causes in AspectJ programs by narrowing down the failure-inducing changes set, with an acceptable runtime performance.

The remainder of this paper is organized as follows. Section 2 introduces briefly the background of AspectJ and gives a motivating example of our approach. Section 3 presents a summary of atomic changes used in fault locating process and gives the semantic inter-relationship rules between those changes. Section 4 reports the implementation issues of our tool Flota and an empirical evaluation is presented in Section 5. The related work and concluding remarks is given in Section 6 and Section 7, respectively.

2. BACKGROUND AND EXAMPLE

Figure 1 and 2 show a small AspectJ example program containing classes `BaseFee`, `TaxFee`, and `ExtraFee`, and aspect `PositiveFeeChecker`. Associated with the program is three JUnit [4] tests, `TestFees.testBase`, `TestFees.testTax` and `TestFees.testExtra` shown in Figure 3. Here, we assume a sequence of edits to the original program in Figure 1 and 2. The editing parts are all new added and marked by underline. Before we present an overview of our approach, we next briefly introduce the background of AspectJ and the change impact analysis technique for

AspectJ programs, which is the foundation of our fault localization approach.

2.1 AspectJ

A *join point* in AspectJ is a well-defined point in the execution that can be monitored - e.g., a call to a method, method body execution, etc. For a particular join point, the textual part of the program executed during the time span of the join point is called the *shadow* of the join point [6].

Sets of join points may be represented by *pointcuts*, implying that such sets may crosscut the system. Pointcuts can be composed and new pointcut designators can be defined according to these combinations. AspectJ defines several primitive pointcut designators; each one is either static (defining a set of join point shadows, such as *call* and *execution*) or dynamic (defining a running condition, such as *cflow*). A combined pointcut is dynamic if one of its component pointcuts is dynamic; otherwise it is static.

Example: In Figure 2, pointcut `singleFeeCheck` contains join points when `BaseFee.calculateNum()` is executed if the runtime this object type is `BaseFee`. Similarly, pointcut `taxFeeCheck` contains join points where `TaxFee.calculateNumm()` is called during execution.

Advice is a method-like mechanism that consists of instructions that execute *before*, *after*, or *around* a pointcut. *Around* advice executes *in place* of the indicated pointcut, allowing a method to be replaced.

An *aspect* is a modular unit of crosscutting implementation in AspectJ. Each aspect encapsulates functionality that crosscuts other classes in a program. Moreover, an aspect can use an *inter-type* construct to introduce methods, attributes, and interface implementation declarations into classes.

Example: The aspect `PositiveFeeChecker` in Figure 2 declares two advices, which are attached to the corresponding pointcut `singleFeeCheck` and `taxFeeCheck`, respectively. These two advices are used to check whether `BaseFee.singleNum` and `TaxFee.taxNum` are in appropriate values during the execution.

An AspectJ program can be divided into two parts: *base code*, which includes classes, interfaces, and other standard Java constructs, and *aspect code*, which implements the crosscutting concerns in the program. For example, the program in Figure 1 and 2 is divided into the base code (shown in Figure 1), which contains three classes `BaseFee`, `TaxFee`, and `ExtraFee`, and the aspect code (shown in Figure 2), which contains one aspect `PositiveFeeChecker`. Moreover, the AspectJ implementation ensures that the aspect and base code run together in a properly coordinated fashion. A key related component is an *aspect weaver*, which ensures that applicable advice runs at appropriate join points. More information about AspectJ can be found in [3].

2.2 Change Impact Analysis

Our fault localization technique relies on the change impact analysis [23] of AspectJ programs. Change impact analysis is performed by Celadon to decompose the source code changes between two program versions into a set of atomic changes. Celadon also builds the semantic dependencies between atomic changes. Flota then uses the output of Celadon, to construct syntactically valid intermediate program versions. These intermediate program versions contain some, but not all of these atomic changes. Notice that, if a set of atomic changes likely contains a bug, then applying certain subsets of that changes will not lead to a buggy program. Thus, the construction of intermediate programs allows us to localize faults more effectively by ignoring the irrelevant changes, focusing our

```

public class BaseFee {
    public int totalNum;
    public int singleNum;
    public BaseFee(int singleNum) {
        this.singleNum = singleNum;
    }
    public int calculateNum() {
        totalNum = singleNum*12;
        return totalNum;
    }
    public int getTotalNum() {
        return calculateNum();
    }
}

public class TaxFee extends BaseFee {
    public int taxNum;
    public TaxFee(int num, int taxNum) {
        super(num);
        this.taxNum = taxNum;
    }
    public int calculateNum() {
        totalNum = super.calculateNum() + taxNum;
        return totalNum;
    }
}

public class ExtraFee extends TaxFee {
    private final int extraNum = 10;
    public ExtraFee(int num, int taxNum) {
        super(num, taxNum);
    }
    public int calculateNum() {
        totalNum = super.calculateNum() + extraNum;
        return totalNum;
    }
}

```

Figure 1: Original version of example program: the base code

attention on viable, interesting ones.

Figure 4 shows the atomic changes corresponding to the source edits in Figure 1 and 2. Each atomic change is shown as a box, where the top half of the box shows the category of the change, and the bottom half shows the method, field or advice involved. An arrow from an atomic change A_1 to A_2 indicates that A_2 is dependent on A_1 .

Example: In Figure 4, the addition of advice: `before(BaseFee tax): taxcheck(tax)` is represented by atomic change 6 (**AEA**: *Add Empty Advice*), which depends on atomic change 5 (**ANP**: *Add new Pointcut*) because the new added advice uses the pointcut declaration `taxFeeChecker`. Atomic change 6 (**AEA**) would lead to a syntactically invalid program unless the referred pointcut is also added (i.e., atomic change 5). Therefore, atomic change 5 is a prerequisite of atomic change 6. The careful reader may also find that there is a **CAB** change (atomic change 8) which depends on atomic change 6. This is because in our change impact analysis model, we decompose the source code editing of adding a new advice into two steps: the addition of an empty advice (i.e., atomic change 6: **AEA**), and the insertion of the advice body (i.e., atomic change 8: **CAB**), where the later is dependent on the former. Similarly, the deletion cases are in a reverse order.

Atomic change **AIC** captures the *advice invocations* changes. It reflects the semantic differences between the original program and the edited program; and indicates that the advice invoking at the certain join points has been changed. The **AIC** changes are generated in situations where `<advice, join point>` pairs is added or removed as a result of source code changes.

Example: Atomic change 9 (**AIC**: `before():taxFeeChecker(), TaxFee.calculateNum()>`) models the fact that the advice `before():taxFeeChecker()` will be invoked at the method call join point `TaxFee.calculateNum()` after editing. As a result,

```

public aspect PositiveFeeChecker {
    pointcut singleFeeCheck(BaseFee base) :
        execution(* BaseFee.calculateNum()) && this(base);
    before(BaseFee base) :singleFeeCheck(base) {
        if(base.singleNum < 0) {
            base.singleNum = 0;
        }
    }
    pointcut taxFeeCheck(BaseFee tax) :
        execution(* TaxFee.calculateNum()) && this(tax);
    before(BaseFee tax):taxFeeCheck(tax) {
        if(((TaxFee)tax).taxNum < 10) {
            ((TaxFee)tax).taxNum = 10;
        }
    }
}

```

Figure 2: Original version of example program: the aspect code

```

public class TestFees extends TestCase {
    public void testBaseFee() {
        BaseFee fee = new BaseFee(100);
        int totalNum = fee.getTotalNum();
        assertTrue(totalNum == 1200);
    }
    public void testTaxFee() {
        BaseFee fee = new TaxFee(50, 20);
        int totalNum = fee.getTotalNum();
        assertTrue(totalNum == 620);
    }
    public void testExtraFee() {
        BaseFee fee = new ExtraFee(0, 10);
        int totalNum = fee.getTotalNum();
        assertTrue(totalNum == 10);
    }
}

```

Figure 3: Test Case for motivating example

the runtime behavior of method `TaxFee.calculateNum()` may be affected by advice `before(): taxFeeChecker()`.

Another core part of our change impact analysis approach is the call graph representation [13,23] for AspectJ programs. Call graph is constructed to determine: (1) the affected tests after program changes; and (2) the affecting atomic changes for each affected test.

Figure 5 shows the dynamic call graphs² for three tests `TestFees.testBaseFee`, `TestFees.testTaxFee`, and `TestFees.testExtraFee`. In these call graphs, edges corresponding to advice invocation is labelled with dash lines. A test is also determined to be affected if 1) its call graph contains a node that corresponds to a base code change [16], like a changed method (**CM**) or deleted method (**DM**) or contains an edge that corresponds to a lookup change **LC**, and 2) its call graph contains a node that corresponds to an *advice body change* (**CAB**), *delete empty advice change* (**DEA**), *modify inter-type method body* (**CIMB**), or *delete the inter-type method* (**DIM**) or contains an edge that corresponds to an advice invocation change **AIC** or contains a node involved in an **AIC** change. Using the call graphs in Figure 5, it is easy to see 1) `Tests.testBaseFee` is not affected and 2) `Tests.testExtraFee` and `Tests.testTaxFee` are affected, because their call graphs each contains a node for `TaxFee.calculateNum()` which involves in the **AIC** atomic change 9.

For the affected tests (`TestFees.testTaxFee` and `TestFees.testExtraFee`), their call graphs on the updated version are shown in Figure 6. The set of atomic changes that affect a given test includes: (1) atomic changes occurred in the base code, changes

²Celadon can work with call graphs constructed using static analysis or from the actual program execution.

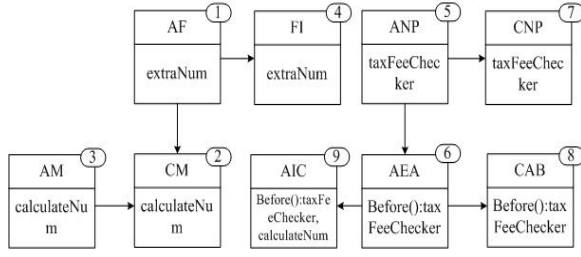


Figure 4: Atomic changes for the example program, with their dependencies

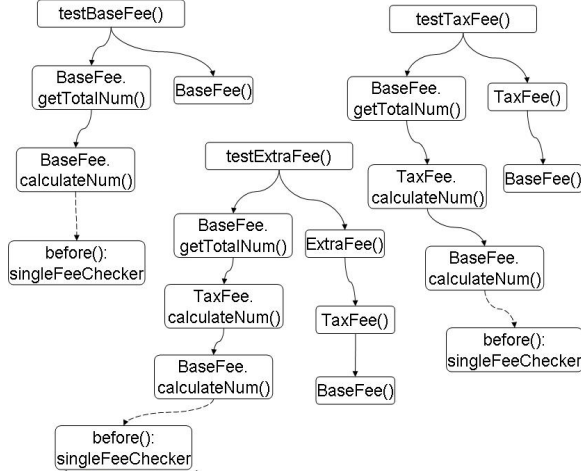


Figure 5: Call graphs for the tests in the original programs

like changed methods (CM) and added methods (AM) that correspond to a node in the edited call graph, and changes like lookup change(LC) that correspond to an edge in the call graph; and (2) the atomic changes appearing in the aspect code, including changes of adding new advice (AEA), changing advice body (CAB), introducing new inter-type declared method (INM) and changing inter-typed declared method body (CIMB) that correspond to a node in the call graph. The affecting atomic changes also include the advice invocation changes (AIC) that correspond to an edge in the call graph. The whole affecting atomic change set also includes the transitively prerequisite atomic changes of all above changes.

Example: We use shadows to annotate the modified method or advice. The call graph of `TestFees.testTaxFee` contains and node corresponding to atomic change 8 and an edge labelled `<before():taxFeeCheck,TaxFee.calculateNum(>`, which corresponds to the atomic change 9 in Figure 4. Atomic change 9 depends on atomic change 5 and 6. Therefore, `TestFees.testTaxFee` is affected by atomic changes 5, 6, 8 and 9. Similarly, the call graph of `TestFees.testExtraFee` contains method `ExtraFee.calculateNum()` corresponding to atomic change 2 which depends atomic changes 1 and 3, and an edge corresponding to the atomic change 9 which depends on change 5 and 6. Consequently, `testExtraFee.calculateNum()` is affected by atomic changes 1, 2, 3, 5, 6, 8 and 9.

2.3 Fault Localization Approach Overview

The original program version passes all the three tests in Figure 3, but test `testFees.testExtraFee` fails after the source editing. As shown in Figure 4, there are totally 9 atomic changes

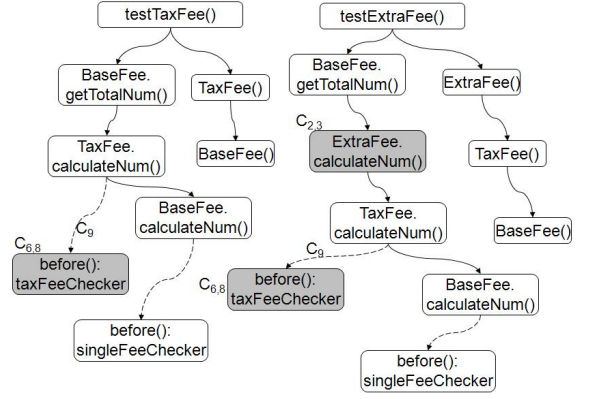


Figure 6: Call graphs for the tests in the updated programs (the call graph of `testBaseFee` remains unchanged after modifications)

from the source editing, 7 changes are identified to affect `testFees.testExtraFee` by Celadon. The question is: *which of those 7 changes are the likely reason(s) for the test failure?* Our tool Flota provides programmers fault localization help by allowing automatic construction of valid intermediate program versions containing certain suspected atomic changes. Flota works as follows: first, a programmer selects the suspected atomic changes that he thinks is the most likely failure reasons, then Flota automatically constructs an intermediate program version by applying the selected changes and all their prerequisites to build a syntactically correct program. The programmer can then re-execute the failed tests against the intermediate program versions. If the test cases pass, the programmer can ignore the applied changes that do not result in the failure, and continues to narrow down the remaining smaller set of changes until they locate the exactly reasons.

Flota also provides programmers a *rollback* function that allows them to undo their selection, and restore the original program. Therefore, programmers can construct the intermediate versions iteratively by applying the suspected changes.

Example: For `TestFees.testExtraFee`, there are 7 atomic changes that may be responsible for the test failure. Programmer may first guess the new added advice invocation is cause of test failure and then he selects change 9 and 8 to apply. Flota automatically applies atomic changes 5 and 6 prior to applying atomic change 9 and 8 to the original program to construct a intermediate program version, which is shown in Figure 7.

```
pointcut taxFeeCheck(BaseFee tax) :
    execution(* TaxFee.calculateNum()) && this(tax);
before(BaseFee tax):taxFeeCheck(tax) {
    if(((TaxFee)tax).taxNum < 10) {
        ((TaxFee)tax).taxNum = 10;
    }
}
```

Figure 7: Intermediate program version by applying atomic change 8 and 9

When programmer re-executes `Tests.testExtraFee` and finds it passes, he ignores the three selected changes and focuses on the last remaining 3 atomic change.

Though a toy motivating example, the debugging support provided by our technique can be useful to real world software and the benefits of having tools such as Celadon and Flota to assist in locating faulty changes are undeniable (discussed in Section 5), especially when there are hundreds or thousands of changes. In our

Abbreviation	Atomic Change Name
AA	Add an Empty Aspect
DA	Delete an Empty Aspect
INF	Introduce a New Field
DIF	Delete an Introduced Field
CIFI	Change an Introduced Field Initializer
INM	Introduce a New Method
DIM	Delete an Introduced Method
CIMB	Change an Introduced Method Body
AEA	Add an Empty Advice
DEA	Delete an Empty Advice
CAB	Change an Advice Body
ANP	Add a New Pointcut
CPB	Change a Pointcut Body
DPC	Delete a Pointcut
AHD	Add a Hierarchy Declaration
DHD	Delete a Hierarchy Declaration
AAP	Add an Aspect Precedence
DAP	Delete an Aspect Precedence
ASED	Add a Soften Exception Declaration
DSED	Delete a Soften Exception Declaration
AIC	Advice Invocation Change

Table 1: A catalog of atomic changes in AspectJ

approach, the syntactic dependence between each atomic change is calculated automatically and programmers only need to focus on the valid, interesting intermediate program versions.

3. ATOMIC CHANGES IN ASPECTJ

In our previous work [23], we identified a catalog of atomic changes (shown in Table 1) and presented a change impact analysis model for AspectJ programs. Those atomic changes represent the source code modifications at a coarse-grained model (that is, method-level), which is amenable to analysis. We assume that the original and the updated programs to be both syntactically correct and compilable.

Additionally, there are syntactic dependencies between atomic changes. Intuitively, an atomic change C_1 is dependent on another atomic change C_2 , if applying C_1 to the original version of the program without also applying C_2 causes a syntactically invalid program that contains some, but not all of the atomic changes. The syntactic dependence relationship (i.e., in above example, C_2 is a *prerequisite* for C_1 , $C_2 \preceq C_1$) between atomic changes is crucial to construct intermediate program versions. However, it is important to understand that only *syntactic* dependencies can not capture all *semantic* dependencies between changes in AspectJ programs (e.g., such as the non-local affecting changes, like adding a new method would cause the advice invoke changes, because of the accidentally join points matching). We will give a suite of dependence rules between atomic changes in Section 3.2 in detail.

3.1 Atomic Changes

Atomic change representation is the foundation of our fault localization technique. Most of the atomic changes such as **CIMB** (change inter-type declared method body regardless of the number of statements within the changed method), **CPB** (change pointcut body) in Table 1 is self-explanatory except for **AIC**. As defined in [23], the formal definition of **AIC** is:

AIC =

$$\{ \langle j, a \rangle \mid \langle j, a \rangle \in ((J' \times A' - J \times A) \cup (J \times A - J' \times A')) \}$$

where J and A are the sets of join points and advices in the original program, and J' and A' are the sets of join point and advices in the modified program. $J \times A$ denotes the matched join points and

advice tuple set in the original program while $J' \times A'$ denotes the matched tuple set in the updated program version.

3.2 Semantic Dependence Rules

There are some inter-relationships between atomic changes which induce a partial ordering \prec on a set of them, with transitive closure \preceq^* . That is, $C_1 \preceq^* C_2$ denotes that C_1 is a prerequisite for C_2 . This partial ordering indicates that when applying one atomic change to the program, all its dependent changes should also be applied in order to obtain a syntactically valid version. The benefit of defining such a partial ordering is that Flota can construct a semantic correctness intermediate version automatically according to the dependence rules.

For a given set C of atomic changes that transform original program P to P' , \prec can be used to determine consistent subsets C' of C such that applying C' to P would result in a valid program P'' that incorporates some, but not all of the changes in P' . A subset of C' of the full set of atomic changes C is consistent if:

$$\forall a' \in C \text{ that } a' \preceq a, a \in A' \rightarrow a' \in A'$$

The ordering between atomic changes is computed automatically by Celadon. We give the dependence rules between atomic changes as follows.

Rule 1: Declare-Access Rule

A new program element must be declared first before making any changes to its body. Similarly, the program element body must be cleaned before deleting the element declaration.

In our change impact analysis model, all the adding changes (**AA**, **AEA**, **ANP**, **INM** and **INF**) represent adding an empty language element. Similarly, all the deleting changes (**DA**, **DEA**, **DPC**, **DIM** and **DIF**) represent deleting an empty element. For example, defining a new aspect is decomposed into a set of atomic changes: first adding an empty aspect definition, then adding its members such as pointcut, advice and inter-type declarations, and finally inserting the body or initializer of each member. Symmetrically, deleting an aspect is decomposed into reserved steps: first clearing the body or initializer of each member, then deleting the member declarations in the aspect, and finally removing the aspect definition. This rule summarizes the dependence relationship of how new aspectual constructs can be added or removed in the program.

Example: In Figure 2, adding a new advice `before():taxFeeCheck` is decomposed into two atomic changes: c_6 (**AEA**: Add Empty Advice) and c_8 (**CAB**: Change Advice Body). According to this rule, we have the dependence relationship: $c_6 \prec c_8$. In Figure 4, the dependence between pairs of atomic changes (1,4), (2,3) and (5,7) are also generated from this rule.

In AspectJ programs, a special case is that the pointcut definition can be anonymous. We treat the anonymous pointcut as a part of advice declaration, thus any changes to the anonymous pointcut will result in the definition change of associated advice.

Rule 2: Declare-Reference Rule

A program element must be declared first before any other program element can have a reference to it. Similarly, a program element can only be deleted when there is no other program element referring to it.

Generally speaking, *Declare-Reference Rule* captures all the necessary AspectJ element declarations that are required to create a valid intermediate version. Dependencies derived from this rule are intuitive as they involve how new code is added or deleted in the program, especially for the interactive usage of AspectJ elements.

Example: As shown in Figure 4, there is a dependence between atomic change 1 (AF) and 2 (CM), because variable `extraNum` is used inside the body of method `calculateNum()`. Therefore, the variable should be declared first before referred in other method body. We have the dependence: $\mathbf{AF}(\text{extraNum}) \prec \mathbf{CM}(\text{calculateNum}())$. In Figure 4, the dependencies between pair of atomic change (5,6) is also derived from this rule.

A special case of the *Declare-Reference Rule* concerns pointcut and soft exception declarations. AspectJ provides an lexical-based join point selection mechanism to pick up necessary join points in the program. The semantics of AspectJ pointcut [6] does not guarantee the join points (e.g. method call or field access) must already be defined in the program. Therefore, there is no dependence between elements used in the pointcut body and the pointcut definition. For example, for a typical pointcut definition like `pointcut pcName(): call(* C.m()), if method C.m()` has not been declared, the program is still valid. Similarly, for the soft exception declaration like `declare soft: Exception: execution(C.m())`, this statement declaration also does not depend on the method `C.m()` declaration.

Rule 3: Abstract-Aspect Rule

An abstract pointcut must be implemented in all the sub-aspect of an abstract aspect, before it is declared in the abstract aspect. Similarly, the declaration of an abstract pointcut must be deleted before the deletion of its implementation in the sub-aspect.

This rule captures the way how a new abstract pointcut is added or deleted according to the AspectJ programming language semantics.

Example: Assume program P defines an abstract aspect AA with a sub-aspect SA which extends AA , we add an abstract pointcut declaration of `pcA` into AA , and SA provides the implementation of `pcA`. According to the *Abstract-Aspect Rule*, we have the dependence between atomic changes: $\mathbf{ANP}(AS.pcA()) \prec \mathbf{ANP}(AA.pcA())$, which means that the pointcut `pcA()` should be declared in the sub-aspect AS before defined in the abstract aspect AA .

Rule 4: Advice-Invocation Rule

The Advice Invocation Change (AIC) depends on the corresponding source code modification that results in the advice invocation changes.

In our model, we use the **AIC** change to model changes of advice invocation behavior. In AspectJ programs, either changes in base or aspect code may cause lost or additional matching of join points, and then results in accidental advice invocation changes. Moreover, common changes like *renaming class member* in base code or *adding new advice* in aspect code may also cause dramatic changes of advice invocation behavior. Therefore, this *Advice-Invocation Rule* is used to capture the dependence relationship of how the advice invocation change occurs. Another issue concerning the advice invocation changes is the aspect precedence declaration for multiple advices invoked at the same join point. The kind of changes are captured by two atomic changes **AAP** (*Add Aspect Precedence*) and **DAP** (*Delete Aspect Precedence*).

Example: In Figure 4, since the new added advice `before(): taxFeeCheck()` matches the method call `taxFee.calculateNum()`, atomic changes 6 (**AEA**) and 9 (**AIC**) are generated. According to this rule, we represent the dependence relationship between these two changes as $\mathbf{AEA} \prec \mathbf{AIC}$.

Generally, this rule summarizes the reasons of why **AIC** change happens. Changes like (1) addition or deletion of class members (methods or fields), (2) addition or deletion of advice definitions, and (3) changes in the pointcut body, may all result in **AIC** change.

However, an inherent intricacy of AspectJ program is that multiple advice can be invoked at the same join point. Thus, we use different $\langle \text{join point}, \text{advice} \rangle$ tuples to represent these cases. For example, if two new added advices $A1$ and $A2$ are both invoked at the same join point JP . We represent this change by atomic change $\mathbf{AIC}(\langle JP, A1 \rangle, \langle JP, A2 \rangle)$. Another special case is the *dynamic pointcut* in AspectJ programs, such as *cflow*, *if* and *target*. Since a dynamic pointcut that statically matches a shadow could potential not match that shadow at runtime, it is difficult to precisely compute the exactly runtime behavior of advice. In our model, we use the $\langle \text{join point}, \text{advice} \rangle$ matching information from the ajc compiler and conservatively assume that for all dynamic pointcuts will be invoked, whether they match a shadow or not has to be determined at runtime. Through this way, though approximately, we can safely use atomic change **AIC** to capture the source code changes.

Rule 5: Inter-type Method Overriding Rule

LC changes also depends on the corresponding inter-type method declaration overriding the existing method that result in the dynamic dispatch change.

As discussed in [17], **LC** change models changes to the dynamic dispatch behavior of instance method. It can be caused by edits that alter inheritance relations. Changes like addition or deletion of a class, addition or deletion of an overriding method as well as changing the modifier of a class or changing the access control of a method may all result in the **LC** change. In AspectJ programs, the inter-type method declaration may also cause the **LC** change, when the introduced method overrides the existing one in the base code.

This *Inter-type Method Overriding Rule* is used to capture the dependencies introduced by the usage of inter-type method declaration. In this case, the **LC** change depends on the inter-type declaration changes in the aspect code.

Example: Assume we add an extra inter-type method (e.g., `public int TaxFee.getTotalNum()`) to aspect `PositiveFeeCheck`. For this change, an object of `TaxFee` will not resolve to `BaseFee.getTotalNum()`. Celadon will report an **LC** ($\langle \text{TaxFee}, \text{TaxFee.getTotalNum}() \rangle$) change. According to this *Inter-type Method Overriding Rule*, we have dependence: $\mathbf{INM}(\text{TaxFee.calculateNum}()) \prec \mathbf{LC}$.

4. IMPLEMENTATION ISSUES

Celadon is a change impact analysis tool for AspectJ programs which is designed as an Eclipse [2] plugin. It first decomposes the source code differences between two program versions into a set of atomic changes, and then automatically determine the affected program parts and affected tests. Our fault locating tool Flota relies on Celadon for: (1) transform program changes into a set of atomic changes, (2) identifying the affected test cases, and (3) compute affecting changes for each affected test.

Like Celadon, Flota is also designed as an Eclipse plugin. It takes the atomic changes, affected tests and their affecting changes generated by Celadon as input. The main tasks that Flota performs are (1) to gather and order all prerequisites of the affecting changes and present them in a dependence tree format, (2) to create the Self-Contained atomic change set according to the programmer's selection, and (3) to apply the Self-Contained change set to the original program to build a syntactically valid intermediate program version. Besides handling full Java language constructs as described in [16], Flota also handles most of the AspectJ language features.

4.1 Create Self-Contained Atomic Change Set

When programmers select and apply the suspected changes, Flota

first construct a Self-Contained atomic change set. A Self-Contained atomic change set is the set which contains exactly all prerequisites of the selected changes together with their dependencies. The semantic dependence rules defined in Section 3.2 are used by Flota to collect all the necessary prerequisites to form the Self-Contained change set. For dependencies defined in Rule 1 to 5, most of the orderings of dependencies handled by Flota are critical to the process of creating valid intermediate program versions. For example, a inter-type method needs to be deleted prior to the addition of a inter-type method in the same aspect with the same name, but different return type.

4.2 Construct Intermediate Version

Flota uses the API exposed by the AJDT [2] project to manipulate the abstract syntax tree (AST) of AspectJ program. When constructing the intermediate program versions, Flota replaces the corresponding AST node including its child nodes (or inserts new AST node) in old version with the edited AST node from the new AST version. For example, if Flota wants to apply an atomic change CM ($C.m()$) to the original program. It first finds the AST node of method $C.m()$ in the updated program version, then it replace the $C.m()$ node in the original program with the updated one. For some changes like AIC, Flota may replace AST nodes in more than one place. And for the selected change that has prerequisites, Flota will first update the AST nodes of these dependent changes before updating the selected one.

Besides handling the technical issues like *Local and Anonymous Class*³ and *Initialize Block* discussed in [8], a special case in Flota implementation is that the advice declaration is anonymous in AspectJ programs. For this case, you can even define two identical advices in one aspect. Therefore, we assign names for each piece of advice manually according to their declaration sequences in the program. The anonymous pointcut is treated as a part of advice declaration, thus any changes to the anonymous pointcut body will result in the definition change of associated advice.

The current granularity of changes Flota handles is at the method (advice) level, because we believe most regression tests focus on functionality at a method level instead of the inner details of method implementation. On the other hand, both Java and AspectJ language provide numerous constructs below the method level such as inner class (aspect). However, the the AspectJ semantic and naming conventions complicate pinpointing its relative position within the enclosing method. Therefore, the current implementation of Flota does not handle sub method level changes individually. We may consider handling such changes in our future work.

4.3 Flota Interface

Flota provides an interface to display the ordering atomic changes between two program versions in a dependence tree manner. As shown by a screen shot (part of the Eclipse development) in Figure 8, Flota presents a graphical depiction of a dependence tree listing all affecting changes and their prerequisites (a tree node depends on its child node). For example, in Figure 8, the highlighted change AEA depends on the AA and ANP changes, while the ANP change also depends on the AA change.

Flota allows programmers to select one or more atomic changes from the dependence tree to apply. It also provides a roll back function to restore the intermediate program to the original one.

5. EMPIRICAL EVALUATION

³Flota can not handle sub-method-level local classes currently, instead treat them as a CM change.

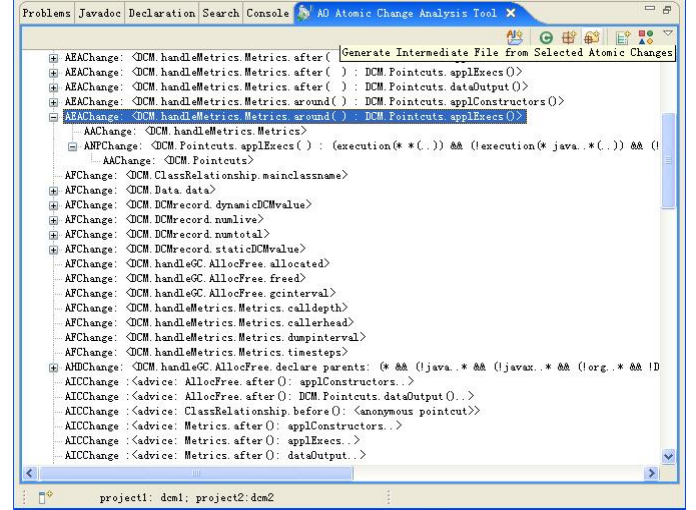


Figure 8: The User Interface of Flota

To evaluate our proposed technique, We performed two case studies on 6 AspectJ benchmark versions using our Flota tool. Next, we will describe our experimental setup, present the case studies, and discuss the results.

5.1 Subjective Programs

Programs	#Loc	#Ver	#Me	#Shad	#Tests	%mc	%asc
Tracing	1059	4	44	32	15	100	100
Dcm	3423	2	249	359	157	94.3	73.5

Table 2: Subject Programs

We use two AspectJ benchmarks for the experimental study. These two programs are included in the AspectJ compiler example package [1]. These two AspectJ benchmarks have also been widely used by other researchers to evaluation their work [9, 10, 21].

Table 2 shows the number of lines of code in the original program (#Loc), the number of versions (#Ver), the number of methods (#Me), the number of shadows (#Shad), the size of the test suite (#Tests), the percentage of methods covered by the test suite (%mc), and the percentage of advice shadows covered by the test suite (%asc). *Advice shadow coverage* is defined as follows. An *Advice shadow interaction* occurs if a test executes an advice whose pointcut statically matches a shadow. The *Advice shadow coverage* is the ratio between *Advice shadow interactions* and the number of shadows in program.

For each program, we made the first version v_1 a pure Java program by removing all aspectual constructs. For some program versions, we made additional modifications to produce more general changes rather than only changes within bodies of methods or advices. We also developed a test suite for each subject program. The experiment was conducted on a DELL C521 PC with AMD Sempron 3.0G HZ CPU and 1.0GM memory.

5.2 Threats to Internal Validity

Although the subjective programs are among the largest available benchmarks, they are smaller than traditional Java software system. For this case, we can not claim that the experiment results can be necessarily generalized to other programs. Other threats mostly concern possible errors in our tool implementations and

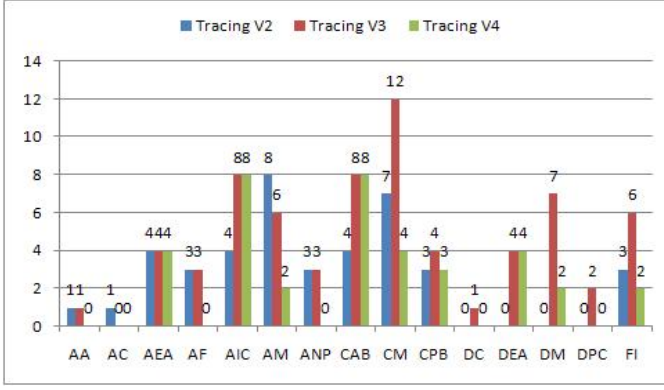


Figure 9: Number of atomic changes between each version pair of Tracing benchmark

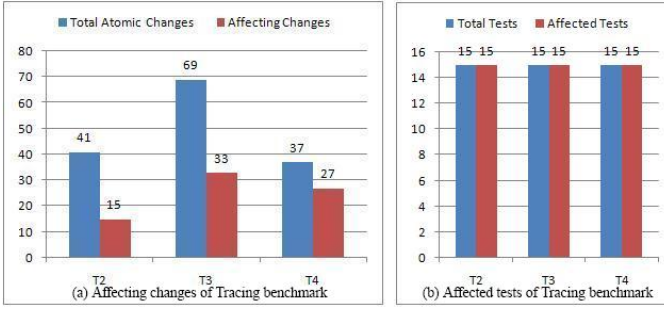


Figure 10: (a) Number of affected changes, affecting changes, and (b) affected tests for each version pair of Tracing benchmark

measurement tools that could affect results. To minimize these threats, we validated the implementations on widely used benchmarks and performed several sanity checks.

5.3 Case Study 1: Tracing

5.3.1 Atomic Changes

Tracing is a small size AspectJ application and its atomic change categories between each successive versions are shown in Figure 9⁴. Note that not every category of atomic change occurs between each version pair. There are total 15 categories of changes among 3 version pairs, in which 8 are aspect-related changes. The most frequent changes of aspect feature is **AIC**, while the overall most frequent change is the **CM**.

5.3.2 Affected Tests and Affecting Changes

The number affecting changes and affected tests are shown in Figure 10. Interestingly, 100% of the tests in each version are affected, because version v_2 adds two pointcuts: `execution(* *())` and `execution(* new())` which cross cut base Java methods and are always executed at runtime. In version v_3 and v_4 of Tracing benchmark, this two pointcuts and their associated advices are all modified. For the affecting changes, in each version, 36.5%, 47.8% and 72.9% of the total atomic changes were responsible for the affected tests.

5.3.3 Debugging Using Flota

⁴The atomic changes between version v_{n-1} and v_n are shown in bar v_n .

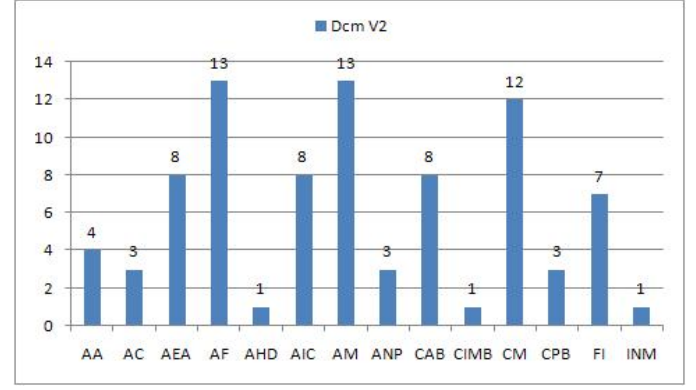


Figure 11: Number of atomic changes between each version pair of Dcm benchmark

It is a challenge to find appropriate test data for Flota to stimulate the debugging activities. However, we found one test, `SquareTest.testPerimeter` which passed in Tracing's 2nd version, but failed in its 3rd version. We use Flota to find the failure-inducing changes of this test. First, Celadon takes the Tracing program version v_2 and v_3 as inputs, and generates atomic changes to represent the source code changes. Second, for the failed test `SquareTest.testPerimeter()`, Celadon lists its affecting atomic changes. Finally, the output of Celadon is passed to Flota.

There are totally 8 affecting changes for the failed test, which account for 13.5% of the total number. First, we selected the **AIC** changes, which is caused by the new added advice `after(): perimeterAndArea()`. This **AIC** change depends on other 4 changes. Therefore, Flota constructs an intermediate program version only containing these 5 changes. We re-execute the failed test cases against this intermediate version and find it passed. Next we select another change **CM** (`Square.square(double, double, double)`) and find it is responsible for the program failure. In order to confirm our result, we apply the other atomic changes to the original version except for **CM** and re-executed `testPerimeter()`, which then succeeded. This indicates that **CM** is the only failure-inducing change.

5.4 Case Study 2: Dcm

5.4.1 Atomic changes

Dcm is one of the largest AspectJ benchmarks available now and its atomic change categories between each version pair are shown in Figure 11. There two versions for the Dcm benchmark. Version v_2 adds a new package **DCM** to the source directory, in which the changes are captured by total 85 atomic changes.

5.4.2 Affected Tests and Affecting Changes

The affecting changes and affected tests are shown in Figure 12. Between Dcm version v_1 and v_2 , about 67.1% of the tests are affected and there are total 85 affecting changes, in which 86.2% is responsible for the affected tests.

5.4.3 Debugging Using Flota

The test suite shown in Table 2 passed in the first version of Dcm benchmark. However, we found one test `DCM.DataTest.testTotalDCM()` failed. We use Flota to locate the exactly failure causes. As mentioned in the first case study, the output of Celadon was passed to Flota and Flota displays the affected tests and its affected changes in a tree format for programmers to select.

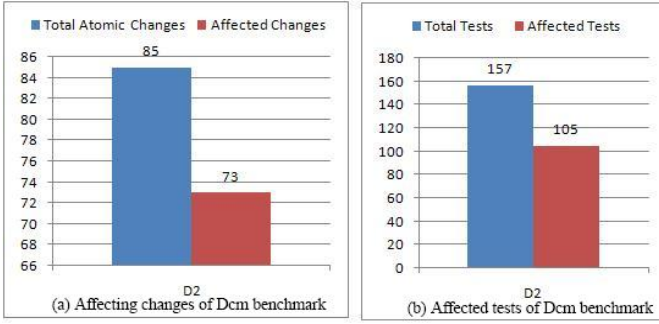


Figure 12: (a) Number of affected changes, affecting changes, and (b) affected tests for each version pair of Dcm benchmark

The test `testTotalDCM()` has 12 affecting changes, which are identified by Celadon from the total of 85 changes between two versions of Dcm. Those 12 affecting changes comprise various kinds of atomic changes, such as **AIC**, **AEA** and **ANP**. Since we are not familiar with the source code, we first guess that the **AEA** (`DCM.handleMetrics.Metrics.after():dataOutput()`) change may alter the program behavior and should be responsible for the failure. We select this change and apply it to the original version. Flota constructs the intermediate program version containing only **AEA** and its 4 depended changes. We test this intermediate version against the failed test and find it passed. Therefore, this 5 changes can be ignored. Next, we suspect another advice change **AEA** (`DCM.handleGC.AllocFree.after():dataOutput()`) to be the responsible one. Similarly, we select and apply this change to construct another intermediate program version. However, this intermediate program version passed the test case again. Therefore, we can ignore those 5 changes further. After two iterations, we focus our attention on two remaining changes **AM** (`DCM.Data.totalDCM()`) and **CM** (`DCM.Data.totalDCM()`), in which the latter depends on the former. We apply the **CM** change to the original version and found the test failed. In order to confirm our result, we apply the other atomic changes to the original version except for **CM** and re-executed `testTotalDCM()`, which then succeeds. This indicates that the change of **CM** (`DCM.Data.totalDCM()`) is the only failure-inducing change.

5.5 Discussion of Experimental Result

In the experiment, we demonstrated the potential ability of Flota to identify the failure-inducing changes in AspectJ programs. During the experiment, programmers can ignore certain changes that do not result in the failure, and narrow down a smaller set of changes until they locate the exactly failure reasons. In our approach, the syntactic dependence between each atomic change is calculated automatically, the Self-Contained atomic change set is computed by Flota and the intermediate program version is guaranteed to be compilable, so the programmers only need to focus on the valid, interesting intermediate program versions. From the experiment, we find Flota can effectively reduce the number of responsible changes when a specific test fails, by repeatedly constructing the intermediate program version. Such as in the Dcm case study, Flota reduced 12 responsible changes to 2 after two iterations, which is much more efficiency than manually inspecting one by one. We also find, in most cases, selecting changes with few or without prerequisites (such as the **CM** change in Dcm case) turns out to be extremely effective to isolate failure-inducing changes.

The method level coarse grained atomic changes simplified the exploration process. However, the programmer may have to con-

sider several related affected tests together with their affecting changes. For AspectJ programs, faults may be caused by multiple source changes, so the exploration of intermediate program versions may also correspond to changes from multiple tests. Therefore, in Flota implementation, we provide a *roll back* function to allow programmer undo the applied changes to ease the process of identifying failure-inducing changes.

6. RELATED WORK

In previous research [23], we described our change impact analysis approach for AspectJ programs and presented our Celadon framework. In [23], we identified a catalog of atomic changes in AspectJ programs and proposed a change impact analysis model to determine affected program parts, affected tests and their affecting changes. In this paper, we describe a fault localization technique for AspectJ programs as an application of change impact analysis. We refine the semantic dependence relationships, which is briefly discussed in [23], between atomic changes, and summarize five dependence rules. These dependence rules is the foundation of creating Self-Contained atomic change set and construct syntactically valid intermediate program versions in Flota implementation.

Other areas of research related to our work are change impact analysis techniques, debugging and fault localization techniques.

6.1 Change Impact Analysis

Recently, many change impact analysis [5, 12, 15, 17, 24] techniques have been proposed in recent years, which are mainly focused on procedural or object-oriented languages. These analysis including static analysis [16], dynamic analysis [12] or, on a combination of the two [5, 15] are concentrated on finding program fragments which are potentially affected by changes. Ryder et al. [17] first use the atomic changes to perform change impact analysis for Java programs. They presented a catalog of atomic changes and a sophisticated definition of dependencies between atomic changes as well as their analysis tool in [16]. However, they focus on the Java language features, and our previous work [23] is an extension of the concept of atomic changes to aspect-related constructs to perform impact analysis for AspectJ programs. In this paper, we proposed a fault localization techniques for AspectJ programs as an application of our change impact analysis approach.

Perhaps the most similar work to ours is the *Crisp* debugging tool for Java programs described in [8]. They use a novel approach to find fault locations in Java programs by isolating the likely failure-inducing changes. Stoerzer [18] et al also presented an change classification tool *JUnit/CIA* built on JUnit and Chianti [16] which classifies Java atomic changes with respect to the tests they affect in order to identify the likely source editing of test failure. And our tool Flota based on the AspectJ change impact model [23] is aimed to provide debugging support for AspectJ programs.

6.2 Fault Localization Techniques

Delta debugging, first proposed by Zeller [22], is used to identify the reasons for a program failure among large sets of textual changes. It searches the entire set of changes and builds the intermediate programs by repeatedly applying different subsets of the changes to the original program. Concerned the differences between program versions, both delta debugging and our work aim at identifying failure-inducing changes. However, Zeller's approach mainly focused on the textual differences like changing one line or one character between succeeding and failing program executions, while our approach and implementation take into account the semantic dependence relationships between small changes (atomic changes) to ensure syntactic correctness.

Other fault localization techniques based on program slicing [20], program dicing [7], thin slicing [14], dynamic hierarchy slicing [19] et al have also been widely proposed by researchers. Computing a slice with respect to an incorrect value determines all statements that may have contributed to that value, and will generally include the statements that contain the error. However, there are several important differences between our approach and slicing's approach to locating faults. Program slicing is a fine grained analysis at the statement level that can be used to inspect a failing program to help locate the cause of the failure, and slicing may become very large. Our work which is at the method level is focused on how to find failure-inducing edits effectively.

7. CONCLUDING REMARKS

In this paper, we presented a fault localization technique for AspectJ programs. The fault localization technique is based on the *atomic change* representation, which captures precisely the semantic differences between two program versions. We also described Flota, a fault localization tool which can be used to isolate failure-inducing changes from others by constructing intermediate program versions. In our experimental study, we found Flota can significantly reduce the number of failure responsible changes and improve the effectiveness of locating faults in AspectJ programs. The empirical data also shows that Flota is very promising in assisting programmers to focus on a very small subset of changes that may alter the behavior of a regression suite.

In our future work, we intend to improve our change impact analysis model described in [23] to capture the semantic changes more accurately in AspectJ programs. We also want to refine the granularity of our atomic change categories to support automatic debugging.

Acknowledgements

This work was supported in part by National High Technology Development Program of China (Grant No. 2006AA01Z158), National Natural Science Foundation of China (NSFC) (Grant No. 60673120), and Shanghai Pujiang Program (Grant No. 07pj14058). We would like to thank Zhongxian Gu, Yu Lin and Chong Shu for their valuable comments and discussion.

8. REFERENCES

- [1] The AspectBench Compiler.
<http://abc.comlab.ox.ac.uk/>.
- [2] AspectJ Development Tools (AJDT).
<http://www.eclipse.org/ajdt/>.
- [3] The AspectJ Team. The AspectJ Programming Guide. Online manual, 2003.
- [4] Junit, Testing Resources for Extreme Programming, 2006.
- [5] T. Apiwattanapong, A. Orso, and M. J. Harrold. Leveraging field data for impact analysis and regression testing. In *Proc. 9th European Software Engineering Conference and 11th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 128–137.
- [6] P. Avgustinov, E. Hajiyev, N. Ongkingco, O. de Moor, D. Sereni, J. Tibble, and M. Verbaere. Semantics of static pointcuts in AspectJ. *Proceedings of the 2007 POPL Conference*, 42(1):11–23, 2007.
- [7] T. Y. Chen and Y. Y. Cheung. On program dicing. *Journal of Software Maintenance*, 9(1):33–46, 1997.
- [8] O. Chesley, X. Ren, and B. G. Ryder. Crisp: A debugging tool for Java programs. In *Proc. International Conference on Software Maintenance (ICSM'2005)*, Budapest, Hungary, September 27–29, 2005.
- [9] B. Dufour, C. Goard, L. Hendren, C. Verbrugge, O. de Moor, and G. Sittampalam. Measuring the dynamic behaviour of AspectJ programs, 2004.
- [10] R. A. Guoqing Xu. Regression test selection for aspectj software. In *In Proc. of the 29th International Conference on Software Engineering (ICSE07)*, pages 65–74, May 2007.
- [11] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proc. 11th European Conference on Object-Oriented Programming*, pages 220–242, 1997.
- [12] J. Law and G. Rothermel. Whole program path-based dynamic impact analysis. In *Proc. 25th International Conference on Software Engineering*, pages 308–318, Washington, DC, USA, 2003. IEEE Computer Society.
- [13] N. Li. The call graph construction for aspect-oriented programs. Master's thesis, School of Software, Shanghai Jiao Tong University, March 2007 (in Chinese).
- [14] S. J. F. Manu Sridharan and R. Bodik. Thin slicing. In *Proc. 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2007.
- [15] A. Orso, T. Apiwattanapong, J. Law, G. Rothermel, and M. Harrold. An empirical comparison of dynamic impact analysis algorithms, 2004.
- [16] X. Ren, F. Shah, F. Tip, B. Ryder, and O. Chesley. Chianti: A tool for change impact analysis of Java programs. In *Proc. Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 2004)*, pages 432–448, Vancouver, BC, Canada, October 26–28, 2004.
- [17] B. G. Ryder and F. Tip. Change impact analysis for object-oriented programs. In *PASTE'01*, pages 46–53, 2001.
- [18] M. Stoerzer, B. G. Ryder, X. Ren, and F. Tip. Finding failure-inducing changes in java programs using change classification. In *SIGSOFT '06/FSE-14: Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 57–68, New York, NY, USA, 2006. ACM Press.
- [19] T. Wang and A. Roychoudhury. Hierarchical dynamic slicing. In *ACM International Symposium on Software Testing and Analysis (ISSTA) 2007*.
- [20] M. Weiser. Program slicing. In *Proc. 5th international conference on Software engineering*, pages 439–449, Piscataway, NJ, USA, 1981.
- [21] T. Xie and J. Zhao. A framework and tool supports for generating test inputs of AspectJ programs. In *AOSD '06: Proceedings of the 5th international conference on Aspect-oriented software development*, pages 190–201, New York, NY, USA, 2006. ACM Press.
- [22] A. Zeller. Yesterday, my program worked. today, it does not. why? In *Proc. 7th European software engineering conference held jointly with the 7th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 253–267, London, UK, 1999. Springer-Verlag.
- [23] S. Zhang and J. Zhao. Change impact analysis for AspectJ programs. Technical Report SJTU-CSE-TR-07-01, Center for Software Engineering, SJTU, Jan 2007.
- [24] J. Zhao. Change impact analysis for aspect-oriented software evolution. In *Proc. 5th International Workshop on Principles of Software Evolution*, pages 108–112, May 2002.